[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 6: Partitioning

Posted by CHARLES *on* 2020-04-09

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

- The main reason for wanting to partition data is **scalability**.
- For very **large datasets**, or very **high throughput**, that is not sufficient: we need to break the data up into **partitions, also known as sharding**.
  - What we call a **partition** here is called a **shard** in MongoDB, Elasticsearch, and SolrCloud; it's known as a **region** in HBase, a **tablet** in Bigtable, a **vnode** in Cassandra and Riak, and a **vBucket** in Couchbase.
- Recently rediscovered by NoSQL databases and Hadoop-based data warehouses.

Partitioning and Replication:

- **Partitioning** is usually combined with **replication** so that copies of each partition are stored on multiple nodes.
- A node may store more than one partition.
  - Each partition's leader is assigned to one node, and its followers are assigned to other nodes.
  - Each node may be the leader for some partitions and a follower for other partitions.

Partitioning of Key-Value Data:

- How do you decide which records to store on which nodes?
- If the partitioning is unfair, so that some partitions have more data or queries than others, we call it **skewed**.
  - A partition with disproportionately high load is called a **hot spot**.
  - Avoid Hotspot:

of knowing which node it is on, so you have to query all nodes in parallel.

- **Partitioning by Key Range**:
  - One way of partitioning is to assign a continuous range of keys (from some minimum to some maximum) to each partition, like the volumes of a paper encyclopedia
  - The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed.
    - In order to distribute the data evenly, the partition boundaries need to adapt to the data.
    - Automatically choose: Bigtable, HBase, RethinkDB and MongoDB(<2.4)
  - Within each partition, we can keep keys in sorted order (see "SSTables and LSM- Trees" on page 76).
- **Partitioning by Hash of Key**: (aka. **Consistent Hashing**)
  - Because of this risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key.
  - A good hash function takes skewed data and makes it uniformly distributed.
    - For partitioning purposes, the hash function need NOT be cryptographically strong: e.g. **Cassandra** and **MongoDB** use **MD5**, and Voldemort uses the FowlerNoll–Vo function
  - Once you have a suitable hash function for keys, you can assign each partition a range of hashes (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition

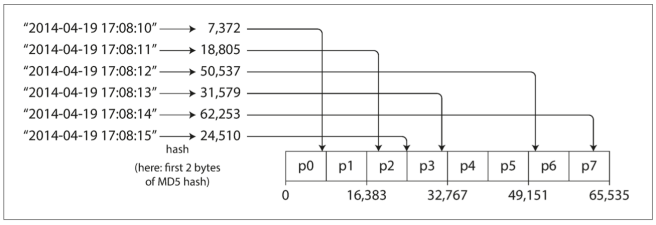

Figure 6-3. Partitioning by hash of key.

    - pseudo-randomly (in which case the technique is sometimes known as **consistent hashing**)
  - **Consistent Hashing**:
    - is a way of evenly distributing load across an internet-wide system of caches such as a content delivery network (CDN).
    - this particular approach actually doesn't work very well for databases [8], so it is rarely used in practice. (C: Seems totally not aligned with what some YouTuber was saying ?)
  - Downside is we lose a nice property of key-range partitioning: the ability to do efficient range queries. (C: this is very similar that when Partition a MQ, the order of the message will not be aligned)
    - In MongoDB, if you have enabled hash-based sharding mode, any range query has to be sent to all partitions.
    - Range queries on the primary key are NOT supported by Riak, Couchbase, or Voldemort.

  - Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in Cassandra's SSTables.
  - The concatenated index approach enables an elegant data model for one-to-many relationships.
- **Skewed Workloads and Relieving Hot Spots**:
  - most data systems are not able to automatically compensate for such a highly skewed workload, so it's the responsibility of the application to reduce the skew.(e.g. Celebrity posts)
    - if one key is known to be very hot, a simple technique is to **add a random number** to the beginning or end of the key.
    - However, having split the writes across different keys, any reads now have to do additional work, as they have to read the data from all 100 keys and combine it.

Partitioning and Secondary Indexes:

- The situation becomes more complicated if secondary indexes are involved.
- A secondary index usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value.
- Secondary indexes are the bread and butter of relational databases, and they are common in document databases too.
- Many key-value stores (such as HBase and Voldemort) have avoided secondary indexes
  - but some (such as Riak) have
- Secondary indexes are the **raison d'être** of search servers such as Solr and Elasticsearch.
- **The problem with secondary indexes is that they don't map neatly to partitions**.
- There are two main approaches to partitioning a database with secondary indexes: **document-based partitioning** and **term-based partitioning**.
- **Partitioning Secondary Indexes by Document**: (e.g. Used car selling site)
  - Widely used: **MongoDB**, Riak, **Cassandra**, Elasticsearch, SolrCloud, and VoltDB.
  - In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition.

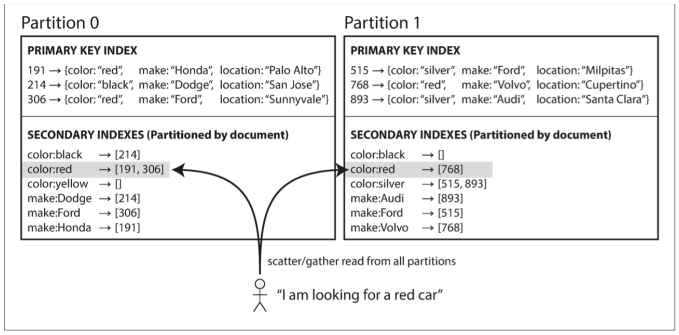

Figure 6-4. Partitioning secondary indexes by document.

get back.
    - This approach to querying a partitioned database is sometimes known as **scatter/ gather**, and it can make read queries on secondary indexes quite expensive. (prone to tail latency amplification)
- **Partitioning Secondary Indexes by Term**:
    - We can construct a **global index** that covers data in all partitions.
    - A global index must also be partitioned, but it can be partitioned differently from the primary key index.
    - We call this kind of index term-partitioned, because the term we're looking for determines the partition of the index.
    - **Pros** it can make reads more efficient:
        - rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants.
    - **Cons** it writes are slower and more complicated,
        - because a write to a single document may now affect multiple partitions of the index (every term in the document might be on a different partition, on a different node).
    - In practice, updates to global secondary indexes are often asynchronous.

Rebalancing Partitions:

- The process of moving load from one node in the cluster to another is called **rebalancing**.
- **Minimum requirements**:
    - the load (data storage, read and write requests) should be shared fairly between the nodes in the cluster
    - While rebalancing is happening, the database should continue accepting reads and writes.
    - No more data than necessary should be moved between nodes. (fast / minimize load)
- **Strategies for Rebalancing**:
    - **How NOT to do it: hash mod N**
        - The problem with the mod N approach is that if the number of nodes N changes, most of the keys will need to be moved from one node to another.
    - **1, Fixed number of partitions**: used in Riak, Elasticsearch, Couchbase, and Voldemort;
        - create many more partitions than there are nodes, and assign several partitions to each node.
            - if a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again.
        - the number of partitions configured at the outset is the maximum number of nodes you can have

partitions, like in the case of a fixed number of partitions.
    - **An advantage** of dynamic partitioning is that the number of partitions adapts to the total data volume.
    - **A caveat** is that an empty database starts off with a single partition, since there is no a priori information about where to draw the partition boundaries. (mitigated by "pre-splitting")
  - **3, Partitioning proportionally to nodes**: used by **Cassandra** and Ketama
    - in other words, to have a fixed number of partitions per node. This approach also keeps the size of each partition fairly stable.
- **Operations: Automatic or Manual Rebalancing**:
  - Couchbase, Riak, and Voldemort generate a suggested partition assignment automatically, but require an administrator to commit it before it takes effect.
  - Fully automated rebalancing can be convenient, but it is unpredictable. (consider rebalancing is a large overhaul), especially dangerous when combined with automatic failure detection.

Request Routing: e.g. ZooKeeper

- When a client wants to make a request, how does it know which node to connect to?
  - This is an instance of a more general problem called **service discovery**, which isn't limited to just databases. (Common problem among "high availability" network applications)
- Few different approaches:
  - Allow clients to contact any node (e.g., via a round-robin load balancer).
  - Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly.
  - Require that clients be aware of the partitioning and the assignment of partitions to nodes.
- How does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?
  - Hard to handle, But many distributed data systems rely on a separate coordination service such as **ZooKeeper** to keep track of this cluster metadata.

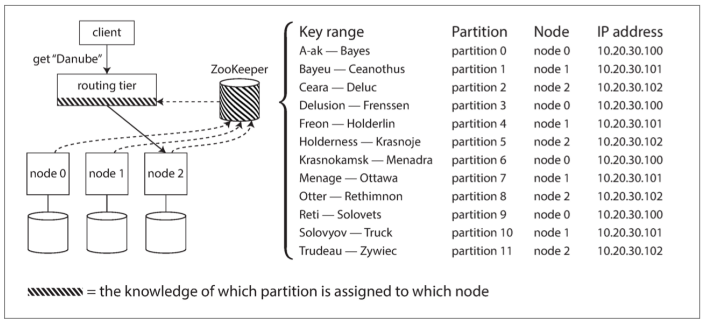

Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

up to date.
    - LinkedIn's Espresso uses Helix for cluster management (which in turn relies on ZooKeeper).
    - HBase, SolrCloud, and **Kafka** also use ZooKeeper to track partition assignment
        - **MongoDB** has a similar architecture
        - (C: Latest version of Kafka seems removed ZooKeeper)
    - **Cassandra** and Riak take a different approach: they use a **gossip protocol** among the nodes to disseminate any changes in cluster state.
        - This model puts more complexity in the database nodes but avoids the dependency on an external coordination service such as ZooKeeper.
    - Couchbase does not rebalance automatically, which simplifies the design.
- **Parallel Query Execution**:
    - **massively parallel processing** (MPP) relational database products, often used for analytics, are much more sophisticated in the types of queries they support.

Summary:

- The goal of partitioning is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load).
- two main approaches to partitioning
    - **Key range partitioning**, where keys are sorted, and a partition owns all the keys from some minimum up to some maximum.
        - partitions are typically rebalanced dynamically by splitting
        - Risk of hot spots
    - **Hash partitioning**, where a hash function is applied to each key, and a partition owns a range of hashes.
        - partitioning by hash, it is common to create a fixed number of partitions in advance
        - Destroy the ordering of keys
- A secondary index also needs to be partitioned:
    - **Document-partitioned indexes (local indexes)**, where the secondary indexes are stored in the same partition as the primary key and value.
    - **Term-partitioned indexes (global indexes)**, where the secondary indexes are partitioned separately, using the indexed values.
- Operations that need to write to several partitions can be difficult to reason about: for example, what happens if the write to one partition succeeds, but another fails?

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>